Java源代码审计字典

目录

Java 源代码审计字典......1
SQL 注入......4
介绍......4
漏洞示例......4
示例一：直接通过拼接 sql......4
示例二：预编译使用有误......5
示例三：%和_（oracle 中模糊查询）问题......5
示例四：order by 问题......6
中间件框架 sql 注入......6
Mybatis 框架......6
介绍......6
漏洞示例......6
示例一：like 语句......6
示例二：in 语句......7
示例三：order by 语句......7
Hibernate 框架......8
介绍......8
漏洞示例......8
审计策略......8
修复方案......8
XSS......9
介绍......9
漏洞示例......9
审计策略......9
修复方案......10
方案一......10
方法二......12
方法三......12
XXE......13
介绍......13
漏洞示例......13
审计策略......13
修复方案......15
XML 问题......16
介绍......16
漏洞示例......16
审计策略......16
修复方案......17
反序列化漏洞......18
介绍......18
漏洞示例......18
示例一 反序列化造成的代码执行......18

示例二 反序列化造成的权限问题......20
示例三 反序列化造成的敏感信息泄露......20
示例四 静态内部类的序列化问题......22
路径安全......23
介绍......23
漏洞示例......23
审计策略......23
修复方案......24
Zip 文件提取......24
介绍......24
漏洞示例......24
审计策略......26
修复方案......26

SQL 注入

介绍

注入攻击的本质，是程序把用户输入的数据当做代码执行。这里有两个关键条件，第一是用户能够控制输入；第二是用户输入的数据被拼接到要执行的代码中从而被执行。sql 注入漏洞则是程序将用户输入数据拼接到了 sql 语句中，从而攻击者即可构造、改变 sql 语义从而进行攻击。

漏洞示例

示例一：直接通过拼接 sql

```
@RequestMapping("/SqlInjection/{id}")
    public ModelAndView SqlInjectTest(@PathVariable String id){
        String mysqldriver = "com.mysql.jdbc.Driver";
        String mysqlurl =
"jdbc:mysql://127.0.0.1:3306/test?user=root&password=123456&useUnicode=true&characterEn
coding=utf8&autoReconnect=true";
        String sql = "select * from user where id=" + id;
        ModelAndView mav = new ModelAndView("test2");
        try{
            Class.forName(mysqldriver);
            Connection conn = DriverManager.getConnection(mysqlurl);
            PreparedStatement pstt = conn.prepareStatement(sql);
ResultSet rs = pstt.executeQuery();
```

审计策略

这种一般可以直接黑盒找到，如果只是代码片段快速扫描可控制的参数或者相关的 sql 关键字查看。

修复方案

见示例三

示例二：预编译使用有误

```
@RequestMapping("/SqlInjection/{id}")
    public ModelAndView SqlInjectTest(@PathVariable String id){
        String mysqldriver = "com.mysql.jdbc.Driver";
        String mysqlurl =
"jdbc:mysql://127.0.0.1:3306/test?user=root&password=123456&useUnicode=true&characterEn
coding=utf8&autoReconnect=true";
        String sql = "select * from user where id= ?";
        ModelAndView mav = new ModelAndView("test2");
        try{
            Class.forName(mysqldriver);
            Connection conn = DriverManager.getConnection(mysqlurl);
            PreparedStatement pstt = conn.prepareStatement(sql);
            //pstt.setObject(1, id); //一般使用有误的是没有用这一句。编码者以为在上面的
sql 语句中直接使用占位符就可以了。常见于新手写的代码中出现。
            ResultSet rs = pstt.executeQuery();
```

审计策略

这种一般可以直接黑盒找到，如果只是代码片段快速扫描可控制的参数或者相关的 sql 关键字查看。查看预编译的完整性，关键函数定位 setObject()、setInt()、setString()、setSQLXML()关联上下文搜索 set* 开头的函数。

修复方案

见示例三

示例三：%和_（oracle 中模糊查询）问题

```
    @RequestMapping("/SqlInjection/{id}")
    public ModelAndView SqlInjectTest(@PathVariable String id){
        String mysqldriver = "com.mysql.jdbc.Driver";
        String mysqlurl =
"jdbc:mysql://127.0.0.1:3306/test?user=root&password=123456&useUnicode=true&characterEn
coding=utf8&autoReconnect=true";
        String sql = "select * from user where id= ?";
        ModelAndView mav = new ModelAndView("test2");
        try{
            Class.forName(mysqldriver);
            Connection conn = DriverManager.getConnection(mysqlurl);
```

```
PreparedStatement pstt = conn.prepareStatement(sql);
pstt.setObject(1, id); //使用预编译
ResultSet rs = pstt.executeQuery();
```

审计策略

定位相关 sql 语句上下文，查看是否有显式过滤机制。

修复方案

上面的代码片段即使这样依然存在 sql 注入，原因是没有手动过滤%。预编译是不能处理这个符号的，所以需要手动过滤，否则会造成慢查询，造成 dos。

示例四：order by 问题

String sql = “Select * from news where title =?” + “order by ‘” + time + “’ asc”

审计策略

定位相关 sql 语句上下文，查看是否有显式过滤机制。

修复方案

类似上面的这种 sql 语句 order by 后面是不能用预编译处理的只能通过拼接处理，所以需要手动过滤。

中间件框架 sql 注入

Mybatis 框架

介绍

漏洞示例

示例一：like 语句

Select * from news where title like ‘%#{title}%’
这样写程序会报错，研发人员将 SQL 查询语句修改如下：
Select * from news where title like ‘%${title}%’
这时候程序将不再报错但是可能会造成 sql 注入。

审计策略

在注解中或者 Mybatis 相关的配置文件中搜索 $ 。然后查看相关 sql 语句上下文环境。

修复方案

采用下面的写法
select * from news where tile like concat(‘%’ ,#{title}, ‘%’)并且上下文环境中手动过滤%

示例二：in 语句

Select * from news where id in (#{id}),
这样写程序会报错，研发人员将 SQL 查询语句修改如下：
Select * from news where id in (${id}),
修改 SQL 语句之后，程序停止报错，但是可能会产生 SQL 注入漏洞。

审计策略

在注解中或者 Mybatis 相关的配置文件中搜索 $ 。然后查看相关 sql 语句上下文环境。

修复方案

采用下面写法
select * from news where id in
<foreach collection="ids" item="item" open="("separator="," close=")">#{item} </foreach>

示例三：order by 语句

Select * from news where title =‘java 代码审计’ order by #{time} asc,
这样写程序会报错，研发人员将 SQL 查询语句修改如下：
Select * from news where title =‘java 代码审计’ order by ${time} asc,
修改之后，程序通过但可能会造成 sql 注入问题

审计策略

在注解中或者 Mybatis 相关的配置文件中搜索 $ 。然后查看相关 sql 语句上下文环境。

修复方案

手动过滤用户的输入。

Hibernate 框架

介绍

漏洞示例

```
session.createQuery("from Book where title like '%" + userInput + "%' and published = true")
```

审计策略

搜索 createQuery()函数，查看与次函数相关的上下文。

修复方案

采用类似如下的方法

方法一

```
Query query=session.createQuery("from User user where user.name=:customername and user:customerage=:age ");
query.setString("customername",name);
query.setInteger("customerage",age);
```

方法二

```
Query query=session.createQuery("from User user where user.name=? and user.age =? ");
query.setString(0,name);
query.setInteger(1,age);
```

方法三

```
String hql=" from User user where user.name=:customername ";
Query query=session.createQuery(hql);
query.setParameter("customername",name,Hibernate.STRING);
```

方法四

```
Customer customer=new Customer();
customer.setName("pansl");
customer.setAge(80);
Query query=session.createQuery("from Customer c where c.name=:name and c.age=:age ");
query.setProperties(customer);
```

XSS

介绍

对于和后端有交互的地方没有做参数的接收和输入输出过滤，导致恶意攻击者可以插入一些恶意的 js 语句来获取应用的敏感信息。

漏洞示例

```
@RequestMapping("/xss")
    public ModelAndView xss(HttpServletRequest request,HttpServletResponse response) throws ServletException,IOException{
      String name = request.getParameter("name");
      ModelAndView mav = new ModelAndView("mmc");
      mav.getModel().put("uname", name);
      return mav;
    }
```

审计策略

扫描所有的 HttpServletRequest 查看相关的上下文环境。

修复方案

方案一

全局编写过滤器

1、首先配置 web.xml，添加如下配置信息：

```
<filter>
	<filter-name>xssAndSqlFilter</filter-name>
	<filter-class>com.cup.cms.web.framework.filter.XssAndSqlFilter</filter-
class>
</filter>
<filter-mapping>
	<filter-name>xssAndSqlFilter</filter-name>
	<url-pattern>*</url-pattern>
</filter-mapping>
```

2、编写过滤器

```
public class XSSFilter implements Filter {
	@Override
	public void init(FilterConfig filterConfig) throws ServletException {
	}
	@Override
	public void destroy() {
	}
	@Override
	public void doFilter(ServletRequest request, ServletResponse response,
FilterChain chain)
		throws IOException, ServletException {
		chain.doFilter(new XSSRequestWrapper((HttpServletRequest)
request), response);
	}
}
```

3、再实现 ServletRequest 的包装类

```
import java.util.regex.Pattern;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletRequestWrapper;
public class XSSRequestWrapper extends HttpServletRequestWrapper {
	public XSSRequestWrapper(HttpServletRequest servletRequest) {
		super(servletRequest);
	}
	@Override
	public String[] getParameterValues(String parameter) {
		String[] values = super.getParameterValues(parameter);
```

```
            if (values == null) {
                    return null;
            }
            int count = values.length;
            String[] encodedValues = new String[count];
            for (int i = 0; i < count; i++) {
                    encodedValues[i] = stripXSS(values[i]);
            }
            return encodedValues;
    }
    @Override
    public String getParameter(String parameter) {
            String value = super.getParameter(parameter);
            return stripXSS(value);
    }
    @Override
    public String getHeader(String name) {
            String value = super.getHeader(name);
            return stripXSS(value);
    }
    private String stripXSS(String value) {
            if (value != null) {
                    // NOTE: It's highly recommended to use the ESAPI library
and uncomment the following line to
                    // avoid encoded attacks.
                    // value = ESAPI.encoder().canonicalize(value);
                    // Avoid null characters
                    value = value.replaceAll("", "");
                    // Avoid anything between script tags
                    Pattern    scriptPattern    =    Pattern.compile("(.*?)",
Pattern.CASE_INSENSITIVE);
                    value = scriptPattern.matcher(value).replaceAll("");
                    //        Avoid        anything        in        a
src="http://www.yihaomen.com/article/java/..." type of expression
                    scriptPattern                                    =
Pattern.compile("src[\r\n]*=[\r\n]*\\\'(.*?)\\\'",  Pattern.CASE_INSENSITIVE  |
Pattern.MULTILINE | Pattern.DOTALL);
                    value = scriptPattern.matcher(value).replaceAll("");
                    scriptPattern                                    =
Pattern.compile("src[\r\n]*=[\r\n]*\\\"(.*?)\\\"",  Pattern.CASE_INSENSITIVE  |
Pattern.MULTILINE | Pattern.DOTALL);
                    value = scriptPattern.matcher(value).replaceAll("");
                    // Remove any lonesome  tag
                    scriptPattern          =          Pattern.compile("",
```

```
Pattern.CASE_INSENSITIVE);
                        value = scriptPattern.matcher(value).replaceAll("");
                        // Remove any lonesome  tag
                        scriptPattern           =           Pattern.compile("",
Pattern.CASE_INSENSITIVE | Pattern.MULTILINE | Pattern.DOTALL);
                        value = scriptPattern.matcher(value).replaceAll("");
                        // Avoid eval(...) expressions
                        scriptPattern    =    Pattern.compile("eval\\((.*?)\\)",
Pattern.CASE_INSENSITIVE | Pattern.MULTILINE | Pattern.DOTALL);
                        value = scriptPattern.matcher(value).replaceAll("");
                        // Avoid expression(...) expressions
                        scriptPattern = Pattern.compile("expression\\((.*?)\\)",
Pattern.CASE_INSENSITIVE | Pattern.MULTILINE | Pattern.DOTALL);
                        value = scriptPattern.matcher(value).replaceAll("");
                        // Avoid javascript:... expressions
                        scriptPattern      =      Pattern.compile("javascript:",
Pattern.CASE_INSENSITIVE);
                        value = scriptPattern.matcher(value).replaceAll("");
                        // Avoid vbscript:... expressions
                        scriptPattern       =       Pattern.compile("vbscript:",
Pattern.CASE_INSENSITIVE);
                        value = scriptPattern.matcher(value).replaceAll("");
                        // Avoid onload= expressions
                        scriptPattern      =      Pattern.compile("onload(.*?)=",
Pattern.CASE_INSENSITIVE | Pattern.MULTILINE | Pattern.DOTALL);
                        value = scriptPattern.matcher(value).replaceAll("");
                }
                return value;
        }
}
```

方法二

首先添加一个 jar 包：commons-lang-2.5.jar ，然后在后台调用这些函数：

StringEscapeUtils.escapeHtml(string);
StringEscapeUtils.escapeJavaScript(string);
StringEscapeUtils.escapeSql(string);

方法三

org.springframework.web.util.HtmlUtils 可以实现 HTML 标签及转义字符之间的转换。

代码如下：

/** HTML 转义 **/

```
String string = HtmlUtils.htmlEscape(userinput);    //转义
String s2 = HtmlUtils.htmlUnescape(string);    //转成原来的
```

XXE

介绍

XML 文档结构包括 XML 声明、DTD 文档类型定义（可选）、文档元素。文档类型定义(DTD)的作用是定义 XML 文档的合法构建模块。DTD 可以在 XML 文档内声明，也可以外部引用。

内部声明 DTD:

引用外部 DTD:

当允许引用外部实体时，恶意攻击者即可构造恶意内容访问服务器资源,如读取 passwd 文件：

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE  replace [
<!ENTITY  test SYSTEM "file:///ect/passwd">]>
<msg>&test;</msg>
```

漏洞示例

此处以 org.dom4j.io.SAXReader 为例,仅展示部分代码片段：

```
String xmldata = request.getParameter("data");
SAXReader sax=new SAXReader();//创建一个 SAXReader 对象
Document document=sax.read(new ByteArrayInputStream(xmldata.getBytes()));//获取 document 对象,如果文档无节点，则会抛出 Exception 提前结束
Element root=document.getRootElement();//获取根节点
List rowList = root.selectNodes("//msg");
Iterator<?> iter1 = rowList.iterator();
if (iter1.hasNext()) {
	Element beanNode = (Element) iter1.next();
	modelMap.put("success",true);
	modelMap.put("resp",beanNode.getTextTrim());
}
...
```

审计策略

XML 解析一般在导入配置、数据传输接口等场景可能会用到，涉及到 XML 文件处理的场景可留意下 XML 解析器是否禁用外部实体，从而判断是否存在 XXE。部分 XML 解析接口如下：

javax.xml.parsers.DocumentBuilder

javax.xml.stream.XMLStreamReader

org.jdom.input.SAXBuilder

org.jdom2.input.SAXBuilder
javax.xml.parsers.SAXParser
org.dom4j.io.SAXReader
org.xml.sax.XMLReader
javax.xml.transform.sax.SAXSource
javax.xml.transform.TransformerFactory
javax.xml.transform.sax.SAXTransformerFactory
javax.xml.validation.SchemaFactory
javax.xml.bind.Unmarshaller
javax.xml.xpath.XPathExpression

XMLInputFactory (a StAX parser)

xmlInputFactory.setProperty(XMLInputFactory.SUPPORT_DTD, false); // This disables DTDs entirely for that factory

xmlInputFactory.setProperty("javax.xml.stream.isSupportingExternalEntities", false); // disable external entities

TransformerFactory

TransformerFactory tf = TransformerFactory.newInstance();
tf.setAttribute(XMLConstants.ACCESS_EXTERNAL_DTD, "");
tf.setAttribute(XMLConstants.ACCESS_EXTERNAL_STYLESHEET, "");

Validator

SchemaFactory factory = SchemaFactory.newInstance("http://www.w3.org/2001/XMLSchema");
Schema schema = factory.newSchema();
Validator validator = schema.newValidator();
validator.setProperty(XMLConstants.ACCESS_EXTERNAL_DTD, "");
validator.setProperty(XMLConstants.ACCESS_EXTERNAL_SCHEMA, "");

SchemaFactory

SchemaFactory factory = SchemaFactory.newInstance("http://www.w3.org/2001/XMLSchema");
factory.setProperty(XMLConstants.ACCESS_EXTERNAL_DTD, "");
factory.setProperty(XMLConstants.ACCESS_EXTERNAL_SCHEMA, "");
Schema schema = factory.newSchema(Source);

SAXTransformerFactory

SAXTransformerFactory sf = SAXTransformerFactory.newInstance();
sf.setAttribute(XMLConstants.ACCESS_EXTERNAL_DTD, "");
sf.setAttribute(XMLConstants.ACCESS_EXTERNAL_STYLESHEET, "");
sf.newXMLFilter(Source);

Note: Use of the following XMLConstants requires JAXP 1.5, which was added to Java in 7u40 and Java 8:

javax.xml.XMLConstants.ACCESS_EXTERNAL_DTD
javax.xml.XMLConstants.ACCESS_EXTERNAL_SCHEMA
javax.xml.XMLConstants.ACCESS_EXTERNAL_STYLESHEET

XMLReader

XMLReader reader = XMLReaderFactory.createXMLReader();
reader.setFeature("http://apache.org/xml/features/disallow-doctype-decl", true);
reader.setFeature("http://apache.org/xml/features/nonvalidating/load-external-dtd", false); // This may not be strictly required as DTDs shouldn't be allowed at all, per previous line.

reader.setFeature("http://xml.org/sax/features/external-general-entities", false);
reader.setFeature("http://xml.org/sax/features/external-parameter-entities", false);

SAXReader

saxReader.setFeature("http://apache.org/xml/features/disallow-doctype-decl", true);
saxReader.setFeature("http://xml.org/sax/features/external-general-entities", false);
saxReader.setFeature("http://xml.org/sax/features/external-parameter-entities", false);

Based on testing, if you are missing one of these, you can still be vulnerable to an XXE attack.

SAXBuilder

SAXBuilder builder = new SAXBuilder();
builder.setFeature("http://apache.org/xml/features/disallow-doctype-decl",true);
builder.setFeature("http://xml.org/sax/features/external-general-entities", false);
builder.setFeature("http://xml.org/sax/features/external-parameter-entities", false);
Document doc = builder.build(new File(fileName));

Unmarshaller

SAXParserFactory spf = SAXParserFactory.newInstance();
spf.setFeature("http://xml.org/sax/features/external-general-entities", false);
spf.setFeature("http://xml.org/sax/features/external-parameter-entities", false);
spf.setFeature("http://apache.org/xml/features/nonvalidating/load-external-dtd", false);
Source xmlSource = new SAXSource(spf.newSAXParser().getXMLReader(), new InputSource(new StringReader(xml)));
JAXBContext jc = JAXBContext.newInstance(Object.class);
Unmarshaller um = jc.createUnmarshaller();
um.unmarshal(xmlSource);

XPathExpression

DocumentBuilderFactory df = DocumentBuilderFactory.newInstance();
df.setAttribute(XMLConstants.ACCESS_EXTERNAL_DTD, "");
df.setAttribute(XMLConstants.ACCESS_EXTERNAL_SCHEMA, "");
DocumentBuilder builder = df.newDocumentBuilder();
String result = new XPathExpression().evaluate(builder.parse(new ByteArrayInputStream(xml.getBytes())));

修复方案

使用 XML 解析器时需要设置其属性，禁止使用外部实体，以上例中 SAXReader 为例，安全的使用方式如下:

sax.setFeature(″http://apache.org/xml/features/disallow-doctype-decl″, true);
sax.setFeature(″http://xml.org/sax/features/external-general-entities″, false);
sax.setFeature(″http://xml.org/sax/features/external-parameter-entities″, false);

其它 XML 解析器的安全使用可参考 OWASP XML External Entity (XXE) Prevention Cheat Sheet

https://www.owasp.org/index.php/XML_External_Entity_(XXE)_Prevention_Cheat_Sheet#Java

XML 问题

介绍

使用不可信数据来构造 XML 会导致 XML 注入漏洞。一个用户，如果他被允许输入结构化的 XML 片段，则他可以在 XML 的数据域中注入 XML 标签来改写目标 XML 文档的结构与内容。XML 解析器会对注入的标签进行识别和解释。

漏洞示例

```
private void createXMLStream(BufferedOutputStream outStream, User user) throws
IOException
{
String xmlString;
xmlString = "<user><role>operator</role><id>" + user.getUserId()
+ "</id><description>" + user.getDescription() +
"</description></user>";
outStream.write(xmlString.getBytes());
outStream.flush();
}
```

某个恶意用户可能会使用下面的字符串作为用户 ID：

"joe</id><role>administrator</role><id>joe"并使用如下正常的输入作为描述字段：

"I want to be an administrator"最终，整个 XML 字符串将变成如下形式：

```
<user>
<role>operator</role>
<id>joe</id>
<role>administrator</role>
<id>joe</id>
<description>I want to be an administrator</description>
</user>
```

由于 SAX 解析器（org.xml.sax and javax.xml.parsers.SAXParser）在解释 XML 文档时会将第二个 role 域的值覆盖前一个 role 域的值，因此导致此用户角色由操作员提升为了管理员。

审计策略

全局搜索如下字符串

StreamSource

XMLConstants

StringReader

在项目中搜索. Xsd 文件

修复方案

```
private void createXMLStream(BufferedOutputStream outStream, User user) throws
IOException
{
if (!Pattern.matches("[_a-bA-B0-9]+", user.getUserId()))
{
}
if (!Pattern.matches("[_a-bA-B0-9]+", user.getDescription()))
{
}
String xmlString = "<user><id>" + user.getUserId()
+ "</id><role>operator</role><description>"
+ user.getDescription() + "</description></user>";
outStream.write(xmlString.getBytes());
outStream.flush();
}
```

这个方法使用白名单的方式对输入进行清理，要求输入的 userId 字段中只能包含字母、数字或者下划线。

```
public static void buidlXML(FileWriter writer, User user) throws IOException
{
Document userDoc = DocumentHelper.createDocument();
Element userElem = userDoc.addElement("user");
Element idElem = userElem.addElement("id");
idElem.setText(user.getUserId());
Element roleElem = userElem.addElement("role");
roleElem.setText("operator");
Element descrElem = userElem.addElement("description");
descrElem.setText(user.getDescription());
XMLWriter output = null;
try
{
OutputFormat format = OutputFormat.createPrettyPrint();
format.setEncoding("UTF-8");
output = new XMLWriter(writer, format);
output.write(userDoc);
output.flush();
}
finally
{
try
{
output.close();
}
```

```
catch (Exception e)
{
}
}
}
```

这个正确示例使用 dom4j 来构建 XML，dom4j 是一个良好定义的、开源的 XML 工具库。Dom4j
将会对文本数据域进行 XML 编码，从而使得 XML 的原始结构和格式免受破坏。

反序列化漏洞

介绍

序列化是让 Java 对象脱离 Java 运行环境的一种手段，可以有效的实现多平台之间的通信、对象持久化存储。只有实现了 Serializable 和 Externalizable 接口的类的对象才能被序列化。
Java 程序使用 ObjectInputStream 对象的 readObject 方法将反序列化数据转换为 java 对象。但当输入的反序列化的数据可被用户控制，那么攻击者即可通过构造恶意输入，让反序列化产生非预期的对象，在此过程中执行构造的任意代码。

漏洞示例

示例一 反序列化造成的代码执行

漏洞代码示例如下：

```
......
//读取输入流,并转换对象
InputStream in=request.getInputStream();
ObjectInputStream ois = new ObjectInputStream(in);
//恢复对象
ois.readObject();
ois.close();
```

上述代码中，程序读取输入流并将其反序列化为对象。此时可查看项目工程中是否引入可利用的 commons-collections 3.1、commons-fileupload 1.3.1 等第三方库，即可构造特定反序列化对象实现任意代码执行。相关三方库及利用工具可参考 ysoserial、marshalsec。

审计策略

HTTP：多平台之间的通信，管理等
RMI：是 Java 的一组拥护开发分布式应用程序的 API，实现了不同操作系统之间程序的方法调用。值得注意的是，RMI 的传输 100%基于反序列化，Java RMI 的默认端口是 1099 端口。

JMX：JMX 是一套标准的代理和服务，用户可以在任何 Java 应用程序中使用这些代理和服务实现管理，中间件软件 WebLogic 的管理页面就是基于 JMX 开发的，而 JBoss 整个系统都基于 JMX 构架。

确定反序列化输入点：首先应找出 readObject 方法调用，在找到之后进行下一步的注入操作。一般可以通过以下方法进行查找：

1)寻找可以利用的“靶点”，即确定调用反序列化函数 readObject 的调用地点。

2)对该应用进行网络行为抓包，寻找序列化数据，如 wireshark,tcpdump 等

注： java 序列化的数据一般会以标记（ac ed 00 05）开头，base64 编码后的特征为 rO0AB。

反序列化操作一般在导入模版文件、网络通信、数据传输、日志格式化存储、对象数据落磁盘或 DB 存储等业务场景,在代码审计时可重点关注一些反序列化操作函数并判断输入是否可控，如下：

```
ObjectInputStream.readObject
ObjectInputStream.readUnshared
XMLDecoder.readObject
Yaml.load
XStream.fromXML
ObjectMapper.readValue
JSON.parseObject
...
```

修复方案

如果可以明确反序列化对象类的则可在反序列化时设置白名单，对于一些只提供接口的库则可使用黑名单设置不允许被反序列化类或者提供设置白名单的接口，可通过 Hook 函数 resolveClass 来校验反序列化的类从而实现白名单校验，示例如下：

```
public class AntObjectInputStream extends ObjectInputStream{
    public AntObjectInputStream(InputStream inputStream)
            throws IOException {
        super(inputStream);
    }

    /**
     * 只允许反序列化 SerialObject class
     */
    @Override
    protected Class<?> resolveClass(ObjectStreamClass desc) throws IOException,
            ClassNotFoundException {
        if (!desc.getName().equals(SerialObject.class.getName())) {
            throw new InvalidClassException(
                    "Unauthorized deserialization attempt",
                    desc.getName());
        }
        return super.resolveClass(desc);
    }
}
```

也可以使用 Apache Commons IO Serialization 包中的 ValidatingObjectInputStream 类的 accept 方法来实现反序列化类白/黑名单控制，如果使用的是第三方库则升级到最新版本。

示例二 反序列化造成的权限问题

Serializable 的类的固有序列化方法包括 readObject，writeObject。
Serializable 的类的固有序列化方法，还包括 readResolve，writeReplace。
它们是为了单例（singleton）类而专门设计的。
根据权限最小化原则，一般情况下这些方法必须被声明为 private void。否则如果 Serializable 的类开放 writeObject 函数为 public 的话，给非受信调用者过高权限，潜在有风险。有些情况下，比如 Serializable 的类是 Extendable，被子类继承了，为了确保子类也能访问方法，那么这些方法必须被声明为 protected，而不是 private。

审计策略

人工搜索文本

```
public * writeObject
public * readObject
public * readResolve
public * writeReplace
```

修复方案

视情况根据上下文而定，比如修改为

```
private void writeObject
private void readObject
protected Object readResolve
protected Object writeReplace
```

示例三 反序列化造成的敏感信息泄露

在 Java 环境中，允许处于不同受信域的组件进行数据通信，从而出现跨受信边界的数据传输。不要序列化未加密的敏感数据；不要允许序列化绕过安全管理器。

```
public class GPSLocation implements Serializable
{
private double x; // sensitive field
private double y; // sensitive field
private String id;// non-sensitive field
// other content
}
public class Coordinates
```

```
{
public static void main(String[] args)
{
FileOutputStream fout = null;
try
{
GPSLocation p = new GPSLocation(5, 2, "northeast");
fout = new FileOutputStream("location.ser");
ObjectOutputStream oout = new ObjectOutputStream(fout);
oout.writeObject(p);
oout.close();
}
catch (Throwable t)
{
// Forward to handler
}
finally
{
if (fout != null)
{
try
{
fout.close();
}
catch (IOException x)
{
// handle error
}
}
}
}
}
```

在这段示例代码中，假定坐标信息是敏感的，那么将其序列化到数据流中使之面临敏感信息泄露与被恶意篡改的风险。

审计策略

要根据实际业务场景定义敏感数据。对于已经被确定为敏感的数据搜索示例一中相关的关键字。

修复方案

上面漏洞示例中的正确写法如下

```
public class GPSLocation implements Serializable
```

```
{
private transient double x; // transient field will not be serialized
private transient double y; // transient field will not be serialized
private String id;
// other content
}
```

要根据实际情况修复。 一般情况下，一旦定位，修复方法是将相关敏感数据声明为 transient，这样程序保证敏感数据从序列化格式中忽略的。

正确示例（ serialPersistentFields ） :

```
public class GPSLocation implements Serializable
{
private double x;
private double y;
private String id;
// sensitive fields x and y are not content in serialPersistentFields
private static final ObjectStreamField[] serialPersistentFields = {new
ObjectStreamField("id", String.class)};
// other content
}
```

该示例通过定义 serialPersistentFields 数组字段来确保敏感字段被排除在序列化之外，除了上述方案，也可以通过自定义 writeObject()、writeReplace()、writeExternal()这些函数，不将包含敏感信息的字段写到序列化字节流中。特殊情况下，正确加密了的敏感数据可以被序列化。

示例四 静态内部类的序列化问题

对非静态内部类的序列化依赖编译器，且随着平台的不同而不同，容易产生错误。

对内部类的序列化会导致外部类的实例也被序列化。这样有可能泄露敏感数据。

```
public class DistributeData implements  SerializedName{
 public class CodeDetail {...}
}
```

CodeDetail 并不会被序列化。

```
public class DistributeData implements  SerializedName{
 public class CodeDetail implements  SerializedName{...}
}
```

报 NotSerializableException，查错误，CodeDetail 这个类虽然实现了 Serializable 接口，但 CodeDetail 在项目中是以内部类的形式定义的。

```
public class DistributeData implements  SerializedName{
 public static class CodeDetail implements  SerializedName{...}
}
```

上面的这种形式可以被序列化但是容易造成敏感信息泄露。

审计策略

人工查找 implements Serializable 的所有内部类

修复方案

```
class ${InnerSer} {}
```
去除内部类的序列化。
```
static class ${InnerSer} implements Serializable {}
```
把内部类声明为静态从而被序列化。但是要注意遵循示例三中的敏感信息问题

路径安全

介绍

不安全的路径获取或者使用会使黑客容易绕过现有的安全防护。
黑客可以改用包含 ../序列的参数来指定位于特定目录之外的文件，从而违反程序安全策略，引发路径遍历漏洞，攻击者可能可以向任意目录上传文件。

漏洞示例

Java 一般路径 getPath()，绝对路径 getAbsolutePath() 和规范路径 getCanonicalPath() 不同。
举例在 workspace 中新建 myTestPathPrj 工程，运行如下代码

```
    public static void testPath() throws Exception{
    File file = new File(”..\\src\\ testPath.txt”);
    System.out.println(file.getAbsolutePath());
    System.out.println(file.getCanonicalPath());
    }
```

得到的结果形如：

```
    E:\workspace\myTestPathPrj\..\src\ testPath.txt
    E:\ workspace\src\ testPath.txt
```

审计策略

查找 getPath，getAbsolutePath。
再排查程序的安全策略配置文件，搜索 permission Java.io.FilePermission 字样和 grant 字样，防止误报。换句话说，如果 IO 方案中已经做出防御。只为程序的绝对路径赋予读写权限，其他目录不赋予读写权限。那么目录系统还是安全的。

修复方案

尽量使用 getCanonicalPath（）。
或者使用安全管理器，或者使用安全配置策略文件。如何配置安全策略文件，和具体使用的 web server 相关。
File.getCanonicalPath()方法，它能在所有的平台上对所有别名、快捷方式以及符号链接进行一致地解析。特殊的文件名，比如“..”会被移除，这样输入在验证之前会被简化成对应的标准形式。当使用标准形式的文件路径来做验证时，攻击者将无法使用../序列来跳出指定目录。

Zip 文件提取

介绍

从 java.util.zip.ZipInputStream 中解压文件时需要小心谨慎。有两个特别的
问题需要避免：一个是提取出的文件标准路径落在解压的目标目录之外，另一个是提取出的文件消耗过多的系统资源。对于前一种情况，攻击者可以从 zip 文件中往用户可访问的任何目录写入任意的数据。对于后一种情况，当资源使用远远大于输入数据所使用的资源的时，就可能会发生拒绝服务的问题。Zip 算法的本性就可能会导致 zip 炸弹（zip bomb）的出现，由于极高的压缩率，即使在解压小文件时，比如 ZIP、GIF，以及 gzip 编码的 HTTP 内容，也可能会导致过度的资源消耗。

漏洞示例

```
static final int BUFFER = 512;
// ...
public final void unzip(String fileName) throws java.io.IOException
{
FileInputStream fis = new FileInputStream(fileName);
ZipInputStream zis = new ZipInputStream(new BufferedInputStream(fis));
ZipEntry entry;
while ((entry = zis.getNextEntry()) != null)
{
System.out.println("Extracting: " + entry);
int count;
byte data[] = new byte[BUFFER];
// Write the files to the disk
FileOutputStream fos = new FileOutputStream(entry.getName());
BufferedOutputStream dest = new BufferedOutputStream(fos, BUFFER);
while ((count = zis.read(data, 0, BUFFER)) != -1)
{
```

```
dest.write(data, 0, count);
}
dest.flush();
dest.close();
zis.closeEntry();
}
zis.close();
}
```

在这个错误示例中，未对解压的文件名做验证，直接将文件名传递给 FileOutputStream 构造器。它也未检查解压文件的资源消耗情况，它允许程序运行到操作完成或者本地资源被耗尽。

```
public static final int BUFFER = 512;
public static final int TOOBIG = 0x6400000; // 100MB
// ...
public final void unzip(String filename) throws java.io.IOException
{
FileInputStream fis = new FileInputStream(filename);
ZipInputStream zis = new ZipInputStream(new BufferedInputStream(fis));
ZipEntry entry;
try
{
while ((entry = zis.getNextEntry()) != null)
{
System.out.println("Extracting: " + entry);
int count;
byte data[] = new byte[BUFFER];
// Write the files to the disk, but only if the file is not insanely
big
if (entry.getSize() > TOOBIG)
{
throw new IllegalStateException(
"File to be unzipped is huge.");
}
if (entry.getSize() == -1)
{
throw new IllegalStateException(
"File to be unzipped might be huge.");
}
FileOutputStream fos = new FileOutputStream(entry.getName());
BufferedOutputStream dest = new BufferedOutputStream(fos,
BUFFER);
while ((count = zis.read(data, 0, BUFFER)) != -1)
{
dest.write(data, 0, count);
}
```

```
dest.flush();
dest.close();
zis.closeEntry();
}
}
finally
{
zis.close();
}
}
```

这个错误示例调用 ZipEntry.getSize()方法在解压提取一个条目之前判断其大小，以试图解决之前的问题。但不幸的是，恶意攻击者可以伪造 ZIP 文件中用来描述解压条目大小的字段，因此，getSize()方法的返回值是不可靠的，本地资源实际仍可能被过度消耗。

审计策略

全局搜索如下关键字或者方法

FileInputStream

ZipInputStream

getSize()

ZipEntry

如果出现 getSize 基本上就需要特别注意了。

修复方案

```
static final int BUFFER = 512;
static final int TOOBIG = 0x6400000; // max size of unzipped data, 100MB
static final int TOOMANY = 1024; // max number of files
// ...
private String sanitzeFileName(String entryName, String intendedDir) throws
IOException
{
File f = new File(intendedDir, entryName);
String canonicalPath = f.getCanonicalPath();
File iD = new File(intendedDir);
String canonicalID = iD.getCanonicalPath();
if (canonicalPath.startsWith(canonicalID))
{
return canonicalPath;
}
else
{
throw new IllegalStateException(
```

```
"File is outside extraction target directory.");
}
}
// ...
public final void unzip(String fileName) throws java.io.IOException
{
FileInputStream fis = new FileInputStream(fileName);
ZipInputStream zis = new ZipInputStream(new BufferedInputStream(fis));
ZipEntry entry;
int entries = 0;
int total = 0;
byte[] data = new byte[BUFFER];
try
{
while ((entry = zis.getNextEntry()) != null)
{
System.out.println("Extracting: " + entry);
int count;
// Write the files to the disk, but ensure that the entryName is valid,
// and that the file is not insanely big
String name = sanitzeFileName(entry.getName(), ".");
FileOutputStream fos = new FileOutputStream(name);
BufferedOutputStream dest = new BufferedOutputStream(fos, BUFFER);
while (total + BUFFER <= TOOBIG && (count = zis.read(data, 0, BUFFER)) !=
-1)
{
dest.write(data, 0, count);
total += count;
}
dest.flush();
dest.close();
zis.closeEntry();
entries++;
if (entries > TOOMANY)
{
throw new IllegalStateException("Too many files to unzip.");
}
if (total > TOOBIG)
{
throw new IllegalStateException(
"File being unzipped is too big.");
}
}
}
```

```
finally
{
zis.close();
}
}
```

在这个正确示例中，代码会在解压每个条目之前对其文件名进行校验。如果某个条目校验不通过，整个解压过程都将会被终止。实际上也可以忽略跳过这个条目，继续后面的解压过程，甚至也可以将这个条目解压到某个安全位置。除了校验文件名，while 循环中的代码会检查从 zip 存档文件中解压出来的每个文件条目的大小。如果一个文件条目太大，此例中是 100MB，则会抛出异常。最后，代码会计算从存档文件中解压出来的文件条目总数，如果超过 1024 个，则会抛出异常。

读者须知：

欢迎加入小密圈【源代码安全审计与 SDL】查看更多代码审计知识，限时低价：